Algorithms, Data Structure & Complexity — Coding Style, Testing, etc.

Uwe Köckemann

January 31, 2020

Abstract

This document is supposed to provide some general hints and tips for programming in C. If you follow these guidelines you will produce better code and save a lot of time when trying to figure out why something is not working.

1 Implementing Algorithms

Consider the following steps whenever you want to implement an algorithm.

1. What are the parameters of the algorithm?
2. What is the exact type of each parameter?
3. What type does the algorithm return?
4. How should these parameters be represented in the language you are using?

   int, double, struct...?
5. Does the algorithm rely on other algorithms?

Example: Consider the dynamic set operations on page 230 in the CLRS book:

```
SEARCH(S,k)
DELETE(S,x)
```

Here, S is the set itself, k is a key value (e.g., int), and x is an element in the set S. When using this to implement a double linked list S represents the list (containing a pointer to the head), x is an element of the list (containing two links and the key), and k is the data (or key) of a list element. Note that whenever a function returns a pointer to an element, you can use that function whenever x is used.

For example to delete an element with value k we can use:

```
DELETE(S,SEARCH(S,k))
```

2 Magic numbers

Avoid magic numbers in your code. Consider the following example:

You need create an array with a fixed size:

```
int a[10];
```

Then later you need to loop over it:

```
for ( int i = 0 ; i < 10 ; i++ )
    ...
```

Changing the size of this array becomes difficult very quickly. To make this more clear, consider using a constant:

```
#define ARRAY_SIZE 10
...
int a[ARRAY_SIZE];
...
for ( int i = 0 ; i < ARRAY_SIZE ; i++ )
    ...
```

(For arrays with sizes that are determined during runtime see Section 8 below.)

3 Naming Things

Make sure your variables, typedefs, etc. use names that refer to what they represent and stick to one way/style of writing names.

```
typedef struct t {
  struct t *next;
  int key;
} b;
```

Here you cannot tell what “struct t” or “b” means without looking at its definition. Using the following will give a more clear idea.

```
typedef struct ListElement_t {
  struct ListElement_t *next;
  int key;
} ListElement;
```

The two names chosen in a typedef struct usually should be almost the same since they represent the same thing. It is usually worth a bit of time to carefully consider names. If you can name things precisely there is a higher chance that you have understood the problem that you have to solve.

4 Global variables

Avoid these as much as possible. Bugs caused by global variables can be hard to figure out because they may change the behavior of functions even if they get the same arguments.

Consider the following example.

```
void divide(int x)
{
  return x / d;
}
```

The outcome of the division depends on the global variable d. If d changes, the result of this function changes even if all arguments are the same. If d is 0 the program will crash but it may be hard to determine where d was set to 0.

5 Arguments of Functions

Data/objects required by functions should always be an argument of that function. Otherwise code-reuse is seriously limited.

Example:

```
void push(int x) {...}
// Where are you pushing?
// What if you need two stacks?
// -> You would need two functions.
```

Better:

```
void push(Stack S, int x) {...}
```

6 When to return NULL

Consider a function

```
int f(int n, int* a) {
  if ( n < 0 ) {
    return NULL;
  }
  return -1 * (a[n]%2) * a[n]*a[n];
}
```

that expects n to be positive and can return any integer. In case n is negative you want to make sure the user of the function knows something went wrong. However, returning NULL as above does not work. Since 0 == NULL the user cannot know if the return value is a regular 0 or NULL. In such cases consider returning a pointer to int to allow the user to test the return value against NULL safely.

```
int* f(int n, int* a) {}
```

Along the same line consider the maximum function for linked lists:

```
int maximum(List *L)
```

If L is empty we would like to return NULL but again it seems to be the same as 0. For this reason its better to return the list element:

```
ListElement* maximum(List *L)
```

Another advantage here is that working with the maximum (e.g., deleting it) becomes much easier when you return a reference to the full object. If you simply return an integer you have to search again for the list element to delete it.

7 Testing

1. Test your code!
2. Create meaningful tests.
3. Don't rely on yourself to parse wrong outputs.
4. Consider testing "sub-functions"

7.1 Meaningful Tests

Make sure to create meaningful tests for each function that you write.

Example 1. *You implemented a stack and test it in the following way.*

```
Stack *S = create_stack();
push(S,1);
push(S,1);
push(S,1);
push(S,1);
push(S,1);
printf("%d", pop(S));
printf("%d", pop(S));
printf("%d", pop(S));
printf("%d", pop(S));
printf("%d", pop(S));
```

This test does not tell us anything about the order in which elements are returned. If push and pop would implement a queue we would see the same result. In fact, if pop returned a random element from S we would not be able to tell the difference to a stack.

7.2 Checking Test Results

Testing a function by printing results is problematic since it requires careful reading of the output every single time. Even if you do it carefully every single time you waste time that could be used better for less boring tasks.

Example 2. *Bad:*

```
printf("%d", myFancyFunction(-2)); // 10
printf("%d", myFancyFunction(10000000)); // -10
printf("%d", myFancyFunction(42)); // 512
printf("%d", myFancyFunction(5)); // 300
printf("%d", myFancyFunction(-42)); // -1
```

This requires reading the output yourself and comparing it to comments in the code.

Instead, since you are programming anyways, you might as well let your code do the work for you.

Example 3. *Better:*

```
int parameter = -2;
int expected_value = 10;
int return_value = myFancyFunction(parameter)
if ( return_value != expected_value )
{
  printf("Test failed: ");
  printf("myFancyFunction(%d) returned %d (expected: %d)",
    parameter, return_value, expected_value);
}
...
```

Once a test is written, running the program will tell you if something went wrong. In addition, this approach only prints in case something went wrong. Less clutter on the screen allows more focus on what's important.

We can take this idea even further by writing a separate main function (i.e., a separate program) for testing.

Example 4. *Write all tests into separate file and put parameters and expected values into an array.*

myFancyFunction.h:

```
int f( int n );
```

myFancyFunction.c:

```
int f( int n )
{
  return 5 + n*n;
}
```

testFancyFunction.c:

```
#include <stdio.h>

#include "myFancyFunction.h"

#define NUM_TESTS 3

int f( int n );

int main( int argc, char* argv[] )
{
  int test[NUM_TESTS][2] = {
    {0,5},
    {1,6},
    {2,9}
  };

  int return_value;

  for ( int i = 0 ; i < NUM_TESTS ; i++ )
  {
    return_value = f(test[i][0]);
    if ( return_value != test[i][1] )
      printf("Test failed: ");
      printf("f(%d) returned %d (expected: %d)\n",
        test[i][0], return_value, test[i][1]);
  }
}
```

Here we have a main function specifically for testing. In this way tests do not fill up the actual program.

7.3 Testing "Sub-functions"

Example 5. *Consider the following function that relies on three other functions* $f1$*,* $f2$*, and* $f3$*.*

```
int f( int n )
{
  int r1 = f1(n):
  int r2 = f2(n);
  int r3 = f3(r1,r2);
  return r3*r3;
}
```

If you write a test for this function and the test fails, the problem may be located in any of the four involved functions. In cases like this it might be useful to write tests also for $f1$, $f2$, and $f3$. If you are reasonably sure that these are fine there is only a very small amount of code left for your bug to hide.

8 Array Structures

When working with arrays consider introducing a new type:

```
typedef struct int_array_t {
  int size;
  int *data;
} IntegerArray;
```

This makes arrays easy to extend and their size is easily accessible.

8.1 Avoiding Array Overflow

Arrays can be extended during runtime. This requires allocating new memory and copying the old array into the new one. A common strategy is to double the current size.

9 Further Reading

A more detailed guide on coding style in C:
`https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html`